# Considerations for the Design of Ada* Reusable Packages

by

Norman S. Nise
Chuck Giffin

Rockwell International
Downey, California
June 4, 1986

## Abstract

This paper discusses two important considerations that precede the design of Ada reusable packages - commonality and programming standards. First, the importance of designing packages to yield widespread commonality is expressed. A means of measuring the degree of applicability of packages both within and across applications areas is presented. Design considerations that will improve commonality are also discussed. Second, considerations for the development of programming standards are set forth. These considerations will lead to standards that will improve the reusability of Ada packages.

## Introduction

By 1990, the cost of software will outpace the cost of hardware by a ratio of five-to-one. According to the United States Department of Defense, the cost of software will rise to $32 billion by 1990, up from $2.5 billion in 1980. The primary responsibility for these high costs can be attributed to the maintenance phase of the software development cycle.

One promising method of reducing these costs and improving the supply is to use what is becoming known as reusable software. Reusable software can be defined to be specifications, designs, data, code, test cases, and documentation that are reused in the same or in a different software program with little or no modifications. Reusability yields a reduction in man-hours required for design, development, testing, and, particularly, maintenance. This reduction in man-hours leads to a reduction in software costs. Since "tried and true" software is used over and over again while bugs are discovered and eradicated, increased reliability is also accrued.

Why hasn't reusable software found widespread acceptance and use by now? The major problem has been the lack of a set of universally accepted standards and a single programming language supporting the design of reusable software. Furthermore, even today, few accept the idea that reusable software could possibly work. Some feel that it is unworkable since a lack of standard and understandable documentation encouraging the use of reusable software exists. Individual company proprietary interests encourage a reluctance to share developed software with other concerns.

* Ada is a registered trademark of the U.S. Government - Ada Joint Program Office

Frequently, others are reluctant to use software developed for other applications since the software would not serve their current needs.

Portability is, of course, another reason for not accepting the concept of reusable software. Code developed on one machine might not run on another without extensive modifications.

Finally, standards used by one company in the development of their software may differ from another company. This lack of standardization makes it difficult to share software with confidence.

What has occurred to change the picture and begin to turn around the lack of community-wide acceptance of the idea of reusable software? First and foremost, the escalating cost of software is driving the change. As pointed out, these costs are rising to unmanageable proportions! This fact drove the Department of Defense to development and declare the use of Ada as the official programming language for mission critical embedded systems. Reusable software can now be a reality for two reasons: (1) a common language, and (2) a language that supports the theoretical basis for reusability.

We now find both government and private industry seriously considering reusable software systems. For example, the Department of Defense Software Technology for Adaptable Reliable Systems (STARS) is currently working with members of private industry to establish criteria for the design of a reusable software system. Such considerations as the library system approach, parts design, metrics, and incentives for participants are being explored. The output from the team will be a reusability guidebook.

The authors have previously described a reusable software system (Reference 11). Commonality was mentioned as a key element for its design. In Reference 12 some design requirements for commonality were described. This paper now ties together both commonality and standards as considerations for the design of reusable software packages.

## Considerations for the Design of Reusable Packages

Regardless of the form that the reusable software system will take, software packages must be designed so that they exhibit certain qualities associated with reusability. If a package is designed with reusability in mind, it will be used again and again. The amount of reuse is a metric that the designer will want to maximize in order to realize the economic advantages of reusable software.

One way of increasing the degree of reuse of software packages is to take specific steps to increase what we call the domain of applicability or the commonality associated with a software package. That is to say, steps must be taken to design software packages that will not only be applicable within a specific applications area, but will also be applicable across applications areas.

Another consideration is to design into the package the basis for reusability and portability. Standards requiring enforcement of these two concepts must be set up a priori to ensure code design that is indeed reusable and portable.

Commonality and standards will now be explored separately to show their importance in the development of reusable software packages.

## Commonality

Commonality is two dimensional. Software reuse can be measured by the degree of applicability of the package both within and across applications areas. Applications areas imply distinct industrial groupings. For example, different applications areas could include missiles, aircraft, spacecraft, weapons, ships, lasers, command/control, radar, etc. The economic advantages of reusable software can be diminished if packages developed for a reusable software system do not have the widest range of applicability. If the designer is satisfied with a very narrow range of applicability, or does not consider extending the range of applicability either within the applications area or across applications areas, the reusable software library will begin to bulge with an overabundance of software from a very narrow domain of applicability. Since each package represents development and maintenance costs, it would be economically beneficial to ensure that the designer develop each reusable Ada package with the maximum possible degree of commonality. Furthermore, the proliferation of packages within the reusable software library could create a problem in classification and retrieval of software.

The space shuttle is an example of the non-reuse of software. Of the millions of lines of code developed, not one line was planned for any reuse on any other project. Hopefully, this will not occur for the software developed for the space station. First of all, a common language, Ada, now makes it feasible to develop reusable software. Second, more sensitivity to the need for creating reusable software now exists. However, what is being suggested here is to take a quantum leap in thinking. To develop reusable software within applications areas is not enough even though it would be a step in the right direction. Reusable software that has had every possible bit of commonality designed into it must be developed. This commonality must cross the boundaries of applications areas if we are indeed to reap the economic benefits of reusable software on a large scale.

Increased commonality needs to be a design consideration up front. The designer must consider how to increase the domain of applicability across applications areas. There must be a reluctance to settle for application-specific packages. For example, a program to add two integers together does not have as wide a domain of applicability as an Ada generic package that provides the choice of variable types.

## A Commonality Matrix

To place the two dimensions that pertain to the domains of applicability into a visual perspective,the commonality matrix is shown on Figure 1.

| WITHIN APPLICATIONS \ ACROSS APPLICATIONS | VERY NARROW | NARROW | WIDE | VERY WIDE |
|---|---|---|---|---|
| VERY NARROW | 0 | 1 | 2 | 3 |
| NARROW | 1 | 2 | 3 | 4 |
| WIDE | 2 | 3 | 4 | 5 |
| VERY WIDE | 3 | 4 | 5 | 6 |

**Figure 1 Commonality Matrix**

The domain of applicability is rated from very narrow to very wide in four steps, both across and within applications areas. Software measured against this matrix has a commonality rating from 0 to 6. The higher the rating, the larger the domain of applicability as measured both across and within applications areas.

Software can be classified within the matrix based upon the expected amount of reuse. In order to estimate this, a detailed domain analysis must be performed to identify the possible users within and across all domains.

The objective of the commonality matrix is to identify a point of departure from which steps can be taken to improve and to expand the domain of applicability as an integral part of the design. The first step is to properly classify the software in order to see the possibility of expanding its domain of applicability. If software is thought of as application-specific, such as spacecraft, aircraft, missile, etc., it will be difficult to think in terms of expanding the software's degree of commonality. However, if functions are thought of rather than applications, the range of possible users enlarges. For example, a sort routine for a spacecraft's downlink data also can be used by the accounting industry. In this case, the mind-set should be focused toward the function "sort" rather than the application "spacecraft". Another reason for doing this is to ensure that the library software's classification does not mask the wide range of applications. The sort routine, classified and buried under a spacecraft application would not be discovered by the accounting industry. In this case, the reuse of one sort routine would be diminished while the library would be expanded by another sort routine from the accounting industry. The economic benefits of the reusable software library will decrease!

The economic benefits that can accrue to an industry taking the time, effort, and money to develop truly reusable packages, can be enormous. A spacecraft industry that has developed reusable sort packages can now market its software products in new applications areas!

As an example, assume that a domain analysis of a navigation function within the spacecraft industry resulted in a commonality rating of very wide applicability. On the other hand, considering the applicability of navigation functions to other areas such as aircraft, accounting, etc., a domain analysis resulted in a rating of narrow. The overall rating for this navigation function as evaluated from the commonality matrix would be 4 which is found at the intersection of very wide within the application area and narrow across applications areas. This type of analysis can then be performed with other functions such as math functions, process functions, mission functions, system outputs, and system inputs, etc.

**Improving Commonality**

The next question that arises is "what can be done to improve the commonality rating of a software package?". A non-reusable package can be thought of as containing application dependent input transformations, application dependent output transformations, and application dependent processes. The package can also contain application independent input and output transformations as well as application independent processes.

One technique would be to create two separate packages. One package would become part of the reusable software library and would contain the application independent input and output transformations as well as the application independent processes and functions. Any transformation or process analyzed to have a narrow range of applicability even within an applications area would be relegated to the non-reusable package. This package would contain application-specific software and would not become part of the reusable software library.

Another technique would be to create an Ada generic package containing the input transformations, output transformations, and processes that have widespread commonality. This package would become part of the reusable software library. An application-specific instantiator would then be written. The function of this package would be to instantiate the Ada generic library package and endow it with all of the application-specific information stored in the instantiator. The instantiator can also be provided with a sequencer in order to instantiate several packages (i.e., input, output, and process).

An example of the first technique is a non-reusable scaler-checker whose function is to take analog and discrete inputs and give messages and scaled data as outputs. The software performs input acquisition, checks for range and limits if the input is analog, checks for desirable states if the input is discrete, scales the inputs, and sends appropriate messages. These functions are supported by a table of ranges, limits, scaling, and messages. By separating the application-specific tables and the conversion to common data types function from the non-application-specific functions performed, a reusable module consisting of range and limit checking, validity checking, and message select functions is formed. The non-reusable module consists of the tables, messages, and conversions to common data types.

## Programming Standards

Another consideration for the design of Ada reusable packages is programming standards. In all software development projects, standards are set, documented, implemented by developers, and audited for compliance by a standards auditing team. Typically, programming standards deal with documentation, naming conventions, restricted language statements, anomalies, interfaces, and the like.

If an Ada reusable software library is to be set up, new standards specifically dealing with the design for reusability must be developed. These standards must exist alongside the standards usually written for software development. Each standard must describe a method of implementation that specifically tells the designer or programmer how to comply. Furthermore, a method of compliance control must exist. Compliance control describes the methods that ensure compliance such as automated techniques or auditing procedures.

Many standards are set up merely as guidelines. Typically these standards are not audited. Other standards are set up as mandatory. They must be followed and automated or audited for compliance.

Reusable software will require both standards that heretofore were not a consideration as well as standards that typically have driven software development in the past.

Naturally, reusable software must be be readable and understandable. To ensure this, the source code must follow prescribed templates so that the user will recognize the same format in all packages. Considerations, such as letter case of types, variables, and subprogram names must be established ahead of time.

Standards for formatting must be in place. The reader must see a familiar format from one reusable package to another. Typically for reusable software, information hiding is a requirement. The method of implementation is hidden from the user. This prevents the user from changing the implementation or becoming confused by it. These standards for reusability must apply to the specification part of the package, the part the user will see. Other standards can be set up to deal with the body. For example, such characteristics as indentation, alignment, and spacing must be written. Comments must accompany all code to improve readability.

Typing and declarations must follow a template. Variables should be in a particular order decided upon a priori. For example, all inputs followed by all outputs.

There are many considerations unique to reusable software. It is beyond the scope of this paper to cover all standards required to build reusability into the software. Some of the important ones to be discussed here are:

(1) Accuracy dependency
(2) range dependency
(3) operation order dependency
(4) side effects

**Accuracy Dependency**

Target machine dependency has an effect upon reusability because of differences in available character sets, differences in exceeding bounds, differences in dynamic allocation and timing effects, the effect upon real-time tasking due to differences in instruction execution time, and differences in accuracy. Factors such as accuracy that affect the portability and reusability of software and should be formalized as programming standards. Floating point operations cannot rely upon the accuracy of the implementation if the code is to be portable. For example, conditional responses cannot rely upon the accuracy of a comparison that can change between implementations. Accuracies must be declared and adhered to. The required accuracy should not exceed that required for a specific application in order to ensure portability to smaller targets. It is a good idea to declare the accuracy of even predefined types to ensure implementation independence. Inequalities using Ada attributes based upon model numbers such as EPSILON, can be used since the same accuracy can be expected with any implementation. Another approach to making "accuracy" implementation independent, is to declare integer and real constants as named numbers of universal type. This leaves it up to the implementation to set the accuracy.

**Range Dependency**

Range constraints for integer and floating point types should be limited so that the ranges will be independent of implementation. This includes integer literals used for discrete ranges. These literals, unless constrained in the declaration could be out of range on some machines.

The use of attributes that are not model numbers, such as FIRST and LAST, should not be used as a range constraints since these attributes are not implementation independent. Furthermore, values of real types that are outside the range of model numbers cannot be handled by every implementation. Thus if these numbers are used for decisions or exception handling, problems will certainly arise.

It is tempting to handle exceptions by using such declarations as NUMERIC_ERROR and CONSTRAINT_ERROR. Unfortunately, the exact conditions causing these exceptions to be raised depend upon the implementation. Reusability would be better served by programming these exceptions directly into the code.

## Order of Evaluation or Elaboration

The order of evaluation of an expression or the order of elaboration of a declaration can be different from implementation to implementation. Standards must be established to ensure that errors do not arise because of differences in the order of elaboration between implementations. The pragma ELABORATE can be used to obtain the same order of elaboration regardless of implementation.

Subexpressions can be evaluated in different ways. Some implementations may evaluate expressions in such a way that causes the subexpression to overflow. Standards must be established to ensure that subexpressions will not overflow under some implementations since range checking cannot be relied upon for intermediate values. One of the ways of accomplishing this task is to limit the number of operators contained in an expression.

Ada generic packages, which will be housed in the reusable library, require special considerations of their own. Code sharing should be avoided. If one package requires code from the other package, the order of compilation will determine if this sharing is possible. Under some implementations this sharing would not even be permitted.

## Side Effects

Another consideration in improving reusability and portability, is the elimination of side effects. Side effects are caused by functions that modify variables which are not local to the expression. A reusability problem arises if these non-local variables are used in the function itself. The reason for the problem is simple. The order of evaluation is essential to create the correct value for the function. Since the order changes between implementations, it is unknown whether the value of the variable used in the function was the one before or after the execution of the function. Establishing standards that set forth the order of variable assignment can prevent the problems associated with side effects. For example, if the right hand side of an expression is completely evaluated prior to the assignment to the left hand side, the previous copy of a variable can be relied upon under all implementations.

## Summary

This paper described two important considerations for the design of Ada reusable packages: (1) commonality and (2) programming standards. It was shown that reusable packages will bring about economic improvement in software development It is imperative that each reusable package be designed to cover the maximum possible domain of applicability. This maximization implies the designing of the package for applications areas outside of that originally intended. Maximizing commonality can be accomplished by thinking in terms of functions rather than applications areas and partitioning application-specific software from the functions that cut across many applications areas. Developers could realize economic gains by extending software sales outside of their own applications area.

Another consideration of this paper was programming standards. It was shown that many standards, previously not required, must be developed to solve the reusability design problem. The areas of concern covered in this paper was the effect of accuracy, range, order of evaluation, and side effects upon reusability. This does not imply that these are the only considerations. This paper attempted to point the way toward new types of programming standards that will be required for the reusable software of the future.

**Acknowledgements**
The authors wish to thank Keith Morris for his invaluable input to this paper.

**References**

1. "Common Ada Missile Packages", Interim Report AFATL-TR-85-17, September 1984-January 1985.

2. Booch, G., "Software Engineering with Ada", The Benjamin Cummings Co., 1983.

3. Freeman, "Reusable Software Engineering:Concepts and Research Directions", ITT Workshop on Reusability in Programming, September 1983

4. Grabow & Noble, "Reusable Software Concepts and Software Development Methodologies", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

5. Honeywell, "RaPIER", Final Scientific Report to the Office of Naval Research, Contract No. N0014-85-C-0666, March 28, 1986.

6. Hughes Aircraft Co., "Reusable Software Implementation Technology Reviews", Prepared for NOSC, December 1984.

7. Jones et al, "Issues in Software Reusability", SigAda.

8. McCain, "A Software Development Methodology for Reusable Components", STARS Workshop 1985 Reports.

9. McCain, "Reusable Software Component Construction: A Product-Oriented Paradigm", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

10. McNicholl & Anderson, "CAMP Preliminary Technical Report", STARS Workshop 1985 Reports.

11. Nise, Dillehunt, McKay, Kim, Giffin, "A Reusable Software System", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

12. Nise & Giffin,"The Design for Reusable Software Commonality", DoD STARS Workshop, March 24, 1986.

13. Nissen & Wallis, "Portability and Style in Ada", Cambridge University Press, 1984.

14. Parnas, D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

15. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, December 1972.

16. Procedings of the Workshop on Reusability in Programming", ITT, September 7-9, 1983.

17. "Reference Manual for the Ada Programming Language", MIL-STD 1851A, 22 January 1983.

18. "Reusable Software", Electrical Design News", February 3, 1983.

19. Snodgrass, "Fundamental Technical Issues of Reusing Mission Critical Application Software", STARS Workshop 1985 Reports.

20. "Strategy for a Software Initiative", Appendix II, Department of Defense,1 October 1982.

21. Van Neste, "Ada Coding Standards and Conventions", Journal of Pascal, Ada, & Modula-2, September/October 1985.

22. Wegner, "Capital-Intensive Software Technology", IEEE Software, July 1984.

23. Witte, B., "Checklist for Ada Math Support Priorities", ACM Ada Letters, March, April 1984.